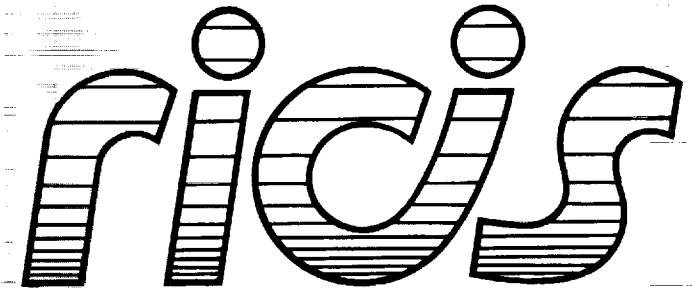
Automating the Verification of Ada Software Development

SofTech, Inc.

9/30/88

Cooperative Agreement NCC 9-16
Research Activity No. SE.13

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

Automating the Verification of Ada Software Development

[redacted] INTENTIONALLY BLANK

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by SofTech, Inc. Dr. Charles McKay served as RICIS research coordinator.

Funding has been provided by the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Stephen A. Gorman, Chief, Systems Support Branch, Flight Data Systems Division, Engineering Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the author and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.

Automating the Verification of Ada Software Development

Prepared by SofTech, Inc., Houston Operations
A UHCL/RICIS Report, Project SE.13
Sept. 30, 1988, SofTech Document HO-004

Introduction

Developing software is a difficult task for people. It requires a precision that is difficult to attain on a regular basis. We can only assume we have written a procedure or other program unit correctly by observing that it compiles and links correctly, that it correctly interfaces with other program units, and ultimately that it executes correctly under the variety of circumstances which it encounters.

Unfortunately, in many cases, the assertion of correctness is essential to the preservation of life and property. The observation of failure in actual use is unacceptable. How can we develop a sense of correctness prior to actual use?

There are two approaches to this task of correctness verification in software systems:

- Formal Verification, and
- Demonstrations of Correctness (testing).

In the first case, formal verification, we apply formal reasoning to the software's textual form. To the extent that it is possible we establish logical assertions about the correct behavior, then apply formal reasoning to the program's code to compare what we've reasoned its behavior will be against what we've asserted it should be.

This process brings two attributes which are reassuring. One is that of formal reasoning, a logical process which is agreed upon and accepted. Independent analysts can agree on the steps and their meaning. The second is that of cross checking between independent sources. The cross checking in this case is between the program code and the formal statement of intended processing. The more independent derivations of what is correct which can be shown to agree with each other the greater the confidence of correctness.

The second approach to verifying correctness is one of demonstrations. To the extent we can mimic the conditions of the software's execution in all cases, we do so and inspect for correct or erroneous execution. This is complicated by a combinatorial explosion of factors which may affect the execution, but provides the powerful reassurance of "having seen it with your own eyes".

Neither of these are perfect, although testing has established itself as the only practical approach in most cases. A degree of testing is necessary in any case to assure the product is correct in its final executable form,

since most formal verification takes place on source code only. Further, formal verification has been considered too cumbersome, error-prone and cerebral for common usage.

This paper discusses the possibilities for changing this situation. While these flaws must be acknowledged today it is also true that the burden of formal verification is lessening and the need is growing. As software systems become larger and more complex it becomes increasingly difficult to test all execution paths and more important that the kernel, critical code be correct. Most importantly, formal verification must be viewed as complementary to testing. It is important to recognize that the path to greater adoption of formal verification need not involve the all or nothing abandonment of testing.

Barriers to Automated Formal Verification

Two of the factors which stand in the way of greater automation of the formal verification process are imprecision in specification and computational magnitude.

For the first of these, the difficulty is that while most computing languages are concise enough to effectively state most computations they are not usually concise enough to support a formal verification of that computation. While this may seem at first glance difficult to accept, it is supported by the recognition that there are many details of the computation which are not of concern in determining the final result but which do lead to some indeterminancy of specific statements in the language. Indeed one of the principles of higher-level programming is the suppression of detail from the statement of the desired computation.

In the case of Ada software, a source of imprecision is the language definition itself. There have been and continue to be efforts at formalizing the definition of Ada, but for the purposes of verification there are some aspects of the language which will by necessity remain too loosely defined. This is one of the limits which is imposed on the potential of formal verification.

The degree of indeterminancy is one of the factors that leads to the second factor, that of the computational load of the verification process. Formal verification is a complex and detailed process which requires both detailed specification and detailed analysis. Without some degree of automation this complexity becomes prohibitive for all but small and critical modules. Even with the automation approaches envisioned for the near future, formal verification will remain an expensive process with an expected application arena consisting of small systems or pieces of systems.

Advancement Paths

The potential for advancement in automating formal verification is based on addressing these barriers. The need for formality is present both in the program code itself and in the assertions of proper behavior. Correspon-

dingly, we need a formal definition of the programming language and what we will call the specification language.

In the case of Ada, there two different approaches which have been adopted: denotational definition and axiomatic definition. Although one may be derived from the other, the axiomatic definition is the form which is preferable for formal verification. Unfortunately, the denotational definition is much further along and has greater support for completion.

In either case, the project is a massive one. The incomplete denotational definition is around 1000 pages. One of its problems is a concern for the accuracy of the work, considering the difficulty of dealing with such a large detailed volume. The cost of doing a detailed formal definition of Ada and in preparing a usable, accurate form for formal verification is perhaps the greatest obstacle to progress in this field. Until this is fully addressed, only limited proofs based on partial and less detailed formalizations of the language can be attempted.

For specification languages it is not yet clear what approach to take, and therefore what language should be used. There are tradeoffs between the characteristics of level of abstraction, expressiveness, formality and ease of use. The English language is one option frequently adopted. However, while it is commonly known, highly expressive and flexible as to the level of abstraction, it suffers greatly from a lack of formality. At the other extreme, a simple annotation of an Ada program which provides a limited ability to assert properties of the program can be considered. In this case the annotations can be made sufficiently formal, but the range of expression is small and the level of abstraction is that of the program itself, not of some higher design principle.

This latter approach has been adopted for some small scale verification systems which attempt to verify limited properties of an Ada program, as will be discussed further. It is also the approach taken by the annotation language ANNA, developed at Stanford University. This language is used with runtime monitoring to provide more substantial support in the verification of applications with tasking in addition to other verification efforts. A more abstract approach is possible through the language Z. This is a mathematically based language based on propositional calculus.

From this consideration of the need for greater formalism it should be clear there is also the need for narrowing the scope of formal verification efforts. In this case, it is important to establish the project priorities and to focus on attainable results. There are important and attainable properties which can be verified without massive computing resources, such as the absence of exceptional conditions (the exceptional condition cannot occur during a normal execution of the program) and lack of infinite loops. These properties are not only useful from a correctness point of view but could also aid in certain program optimizations.

The other aspect of narrowing the scope of verification efforts is be selective in its application. Not all parts of a system are equally critical to the acceptable behavior of the program. There is an analogy here to the development principle of building a system first, then analyzing where it spends most of its time to decide where to concentrate efforts at improving its performance. Not all sub-systems are equally critical to high performance.

As a final note, the advancement in the technology of computing systems (AI and expert systems) and support tools for programmers has made the development of an automatic verification system very possible. What is needed is the recognition of the value of such a system and making it broadly accessible. While there are the difficult aspects of establishing a basis of formality and in dealing with the computational magnitude of formal verification techniques, there are ways of constraining these problems and reducing their scale. With the right support systems, formal verification can be a significant tool for programmers and system developers in the development of reliable, dependable software.